VZDU # 27 NOV/DEC 1990

VZ DOWN UNDER CLUB

Editor
Harry Huggins
12 Thomas Str.
Mitcham. 3132.
03-873 1408

Treasurer
Ron Allen
2 Orlando str.
Hampton.3188.
03-598 4534

Having seen the er... "MARVELS" of the ..er.. "PREMIER STATE", it is quite a relief to return to the "SANITY" of a less boastful locale now the finals fever is over!

Still it was quite interesting to see the "PREMIUM" shortcomings of the adjoining state. In fact the only place I noticed Premium was on their number plates, and that has been deleted from the newer ones. They are learning!

The computer show was quite interesting, and well worth attending. On display was the whole range of computers, from the mighty VZ to the "run of the mill" IBM. The VZ was the only one I noted doing "something new". This was in several fields, such as the Analog joystick, a mouse, speech synth., transmission of programs over phone lines with serial interface and modem, a very accurate capacitance meter and an ohm meter,making use of the VZ's (as yet not fully utilise) powers. These were further demonstrated at the HVVZUG meeting on 5th Oct., at which I was warmly welcomed.

I have been assured that in due course all these will appear in the HVVZUG newsletter, and maybe we will be permitted to copy them.

I do however return with several programs that will interest our next meeting.

The other computers were doing their usual stunts, Databases, Spread Sheets and Wordprocessors. The Commodores and Ataris were playing games at which they are so good. Of note was a T.I. (Texes Instruments) controllig a model train, entirely from a program.

I make mention of David's article, and I also mention details at end of Games Column. It is quite an effort, and we would like to see more like it. How about it members?

Meetings are held on the first Sunday of each month, now at 12 THOMAS STR. MITCHAM. Any time after midday is a good time to turn up.

Last issue, in Letter to the Editor, the Data statements were missed out. You will find them in The Hhackers Column. Sorry about that. I was expecting another letter to editor this month, but it did'nt arrive. Also looking forward to another SCREAM SHEET. The author should be well equiped now to really get down to "DEAR HARRY"

A MERRY CHRISTMAS to you all, and a very satisfactory New YEAR.

Harry

ADVENTURE GAME WRITING FOR THE VZ200/300

By DAVID WOOD

If you have never written an adventure program before, it isn't as hard as it seems. It's true that adventure programs are very long, and take a long time to prepare and type in, but most of the actual program coding isn't all that difficult. The more complicated pieces of programming, like the sections of the program that display room descriptions and contents, or allow the computer to understand what the player types in, don't vary much from program to program, and can be copied when you come to write a program of your own.

There may be some of you that have never played an adventure game before. If you haven't, there are some good programs available on VZ Down Under's Public Domain tape #1, with a few other good adventures on the other tapes. There have also been a few adventure programs published in this magazine - 'The Thief of Baghdad' (VZDU #1), 'Silver Mountain' (VZDU #4 - VZDU #9) and 'Merkfruit lodge' (VZDU #23).

With this is a sample adventure program for you to type in. it is comparatively short, but it still won't quite fit into an unexpanded VZ300. It will fit if the following lines are deleted:
1130,1140,1160,1170,1180.
Unfortunately, when we try to add a few extras to the program, like saving and loading of the players progress in the game so far and longer room descriptions, you won't be able to add these.

The saving/loading routines mentioned above haven't been included at this stage, because the PRINT# routine doesn't work for all types of datasette! The routines used are quite complex and so will be discussed in a separate article. They are based on the memory dump article in VZDU #7, so enthusiastic tape users, or disk users, can add their own routines if they want to.

This program won't be available as public domain software as the second best way to learn how to write an adventure program is to type one in. (The best way to learn is to write one yourself!). It may be made available on a swap basis for other adventures, but I haven't organised anything about this yet.

If you want to enjoy playing this game, type it in fairly soon as we will look at the program in detail in later editions - this may give too many clues on how to solve it! Type the program in very carefully as small bugs (mistakes) can take ages to track down. As soon as you get tired or grumpy, STOP, save your program to tape or disk, and resume typing it in later on. Save the program before running it for the first time, as it has some machine code routines in it, which could make the program crash if there are any mistakes in them.

Lastly, if you have any problems understanding this, or any other of these articles, please write to me at:

RMB 1815
Samaria Rd
Benalla Vic 3673.

I will reply to all correspondence eventually, but this may take some time as I am doing year 12, and homework commitments must come first.
the first real article will appear in the next edition.

There are several ways to design an adventure game. The method described in this and the next few articles is the one I personally use, but this doesn't mean it is the "right" way. If you prefer to use a different method of design, or programming for that matter, then do so.

Firstly think of an object for the game - a setting for it to take place in and some major task for the player to perform. Common goals of adventure games are to collect some treasure, to escape from somewhere, to rescue one or more people, to solve a mystery, to lift a curse, or to save life on earth as we know it from total destruction, just to name a few. The setting can be in the past, the present or the future - in space, in a magical land or an ordinary building. To summarise, the only limit on tasks or settings is your own imagination.

You then think of a character to be controlled by the player. Try to make it fit in with the setting of the game. A barbarian hero would be well suited to a "Sword and Sorcery" type game but would be rather out of place in a space adventure.

If you can't think of a character or setting for your adventure, try stealing one! Books, television and films could be used for this. You can always change the names of them later on when you have got a few more ideas. (If you use the original names, there could be a few copyright problems if you try to sell the game, but it is OK if you only plan to write the game for your own use.) Another approach you could use is to play a game from a different type of machine until you have solved it, then write it for the VZ from scratch. This was quite succesfully done by Scott Le Brun, who converted the game "King's Quest" for the VZ300, by completely rewriting the program code. The result was the program "Knight's Quest", which is considered to be one of the best adventures available for the expanded VZ300. There are more copyright restrictions with this method, and the usual problems with lack of memory space.

Don't waste time trying to think of something if you don't feel like it. Games designed like this aren't usually successful. Wait for a while and you will probably think of something later on.

Make sure the plot of your game is one you are interested in. It takes a long time to get from the design stage to the finished product, so if the game bores you, it is quite likely that you will give up. I have quite a few maps lying around at home, of adventures that never made it off the drawing board.

Once you have thought of a setting, goal and character, you need to think of some other tasks and traps for your player before finishing the game. Completing the major task of the game should be the climax. This part should not be easy - it should require a burst of truly creative thinking. For example, if the object of the game is to destroy a malevolent robot, typing KILL/ SMASH/ DESTROY ROBOT should not work. Instead use something more imaginative.

eg.

    The robot sees you and begins to lumber slowly towards you.
    What next? DROP BANANA PEEL
    The robot has tripped over the banana peel and crashes to the floor.
    You have angered the robot and he is about to struggle to his feet and
    throttle you.
    What next? EXAMINE ROBOT
    There is a small slot in the back of his neck.
    What next? INSERT CARPET FLUFF
    The fluff has blown all his circuits and he now can't move a single
    motor.

To add to this when the player goes through rooms containing carpet fluff or banana peels, he/she is likely to consider them of questionable value and ignore them. Another method would be leave the banana skin around a banana, and the player would be tempted to eat the banana and throw the peel away.

Once you have thought of a few tricks and traps, you should know if you program is going to work or not. If you can't think of anything, it may be that your plot isn't

feasible. You should be able to get some general ideas from playing other adventure games, but never take something directly from another one. If players have played this adventure before, they will know the solution to the problem straight away, which will make the game rather boring. (I have come across at least three adventures where you have to SPRAY bats or insects with an aerosol can.) Likewise, don't repeat the same problem in the one adventure.

Remember not to make the game so long that it won't fit in the available memory or too difficult to program. Don't make the game so complicated that the player has never seen anything like it before. Even though they want the game to be full of new and original puzzles, they want it to be at least partly familiar. (Most people only read a few different types of books, like spy stories or science fiction, but of course they don't want each one to be exactly the same.) Also remember to not be so devious the player has no hope of solving the problem, and, of course, not make the game so easy that it is boring and will be solved in the first few attempts. There is a difference between being obvious and being comprehensible. Many commercial adventures state in their advertising that they will take many weeks to solve, and this is a major selling point. Don't put too many braincrunching problems at the start of the game. Give the player a chance to explore and "become part of the game" first. While difficult problems are definitely needed, if they are placed at a stage before the player is "hooked" into the game, he/she will probably get frustrated and give up.

You may like to place some random events in your program. If you do this, don't use them to kill the player for no apparent reason. Instead use them to produce different secret passwords or combinations for each game, to randomly place an object that the player has to find (or a monster), or to help the player overcome a situation, like falling into a pit, where he/she would otherwise be killed. People who have played "Castle Greystone" would know that they have to kill zombies that appear at random around the castle with weapons that they may find in there. This is quite a reasonable use of random events, except that often zombies appear and kill players before they have had a chance to find a suitable weapon.

If you have thought of a few problems but are stuck for a while, start drawing your adventure map and you will probably think of a few more ideas. When you have finished thinking up your plot, write down all of the object words the computer will need to understand. Sort these into lists of "gettable" objects (Objects the player can / is allowed to pick up and carry) and "ungettable" objects (objects that can't be picked up by the player). This is important because the computer needs to know where any portable object is at any one time.

Some people think that any gettable object should have a use, but if the player realises this, he/she will know that they have to have every object at one stage or another, and this would ruin the banana peel / carpet fluff effect described earlier.

It is better to have some objects having uses, some having negative effects (like a box of gunpowder that explodes and does you grievous bodily harm every time you go into a room with a fire in it), and some that don't have any use at all, except to confuse the player ("How am I supposed to unbolt that useful looking sword from off the wall, and do I really need a broken left handed Farenheit scale thermometer?").

Next make up a list of verbs for the player to use as input. Common verbs include N,S,W,E (these are single letter abbreviations for the direction comands north, south, east and west), HELP (lists all the verbs the computer knows), INV (lists what the player is carrying), GET, LOOK, EXAMINE, OPEN, LEAVE, DROP, UNLOCK,LIGHT, etc. The HELP command is added because the player should have enough problems as it is, without having to work out what word to use. There is very little more frustrating than than knowing exactly what to do, but not knowing what word to use. If the correct words for a situation is "SCALE ROPE" and the player types "CLIMB ROPE", the computer will probably respond with something like "YOU CAN'T DO THAT." The most obvious reaction from the player is to assume that the rope can't be climbed, rather than rush down to the bookshop and buy the latest Thesaurus. The fact is that different writers use different words for the same situation. In two different adventures I have played, one

insisted that you don't PUSH boats - you have to LAUNCH them. The other didn't understand what I meant by "LAUNCH," but responded perfectly when I told it to PUSH the boat. One exception to listing all your words is that if you have any words that would make an action rather obvious, like "VACUUM CARPET", you may prefer not to list them, but make the words fairly usual, and tell the player that some words have been omitted.

You might like to include two words for the same action, so there aren't as many occurrences of "YOU CAN'T DO THAT" or "I DON'T UNDERSTAND". examples of this are "LOOK" and "EXAMINE",or "UNLIGHT" and "EXTINGUISH" (UNLIGHT isn't a real word, but not many people like typing or spelling big long words like EXTINGUISH). You might also like to give them slightly different meanings. "LEAVE" and "DROP" could have the same functions except things that are DROPped are more likely to smash, splatt, grow little green legs and run away, etc. Avoid using words like "USE" or "KILL" - make the player be more specific.

When you come to the stage where you are about to design your map, decide how many rooms you are going to have, and draw up a grid of appropriate size. If you want to have 64 rooms make the grid 8 x 8 squares wide, or if you want 80 rooms make it 8 x 10 squares wide, for example. The number of rooms depends on imagination and memory space. You don't have to use a grid, but make sure there is no more than one exit from a room in any direction - for example don't have a room with two exits to the south, both of which lead to different rooms, or things get rather confusing. (Locations in adventure games are generally referred to as "rooms" even though they may not be - they could be parts of real rooms, somewhere out in the garden or in a forest.) Make each square have side length of 25 to 30 millimetres - an inch to a little bit over an inch for the non-metric minded - although you might need to make them smaller if you want the map to fit on one page. Draw your map in pencil at first because you may make some mistakes, decide that you don't want to use a part that you have already drawn in, or suddenly think of something that would make your game much more interesting, only to find you don't have enough space for it on your map and have to either:-

*throw your old map in the bin and start again.

*leave out the idea.

You then decide where you are going to place your player at the start of the adventure. If it is taking place in a house, a logical starting point is in the garden, in one of the outside squares. Pencil in short descriptions for other rooms next. They don't have to be literary masterpieces - a few words for each room, like "bedroom", "entrance hall", or "room with locked door", will do at this stage. Draw in walls for every exit that the player can't move through. This is usually a double line drawn around the boundary of the map any anywhere else you decide is impenetrable. (Again, these are referred to as "walls", although they could be a tall and thick hedge, a rock fall or a cliff face.) Next place a number in the top left hand corner of each square, moving from left to right, and top to bottom, starting at one, and ending at 64, or 80, or however many squares you happen to have. The reason why we start at one, and not zero, will become apparent when we come to write the program. Many games which start the numbering at zero read the room descriptions from memory into a data array, which means that the room descriptions are then in two different places in the memory at once, which in the case of the VZ is an appalling waste of precious memory. When writing adventure games for the VZ there isn't very much memory to play around with.

Next arrows are drawn in each square for the directions that the player can move whilst in the square, and these are labelled with "N","S","W" or "E". North is usually taken towards the top of the page. Following this you are ready to write down your "movement codes," which will be needed for the program. If your map is a grid system, for each room decide for north, south, west and east (in that order) if there is an exit in that direction, and write down a "0" if there is, and a "1" if there isn't. For example if in one room there are exits to the south and east, but not to the north and west, the movement code for that room will be "1010". Make up a movement code for each room and write them down somewhere. If you aren't using a grid structure, use "00" for any direction in which there is no exit, and the room number of the destination for

directions in which there are exits. (You should use a leading zero for rooms one to nine.) If for example there is a room with no exits to the south or east, an exit north to room 8, and an exit west to room 17 the movement code in this case would be "08001700". This is where the advantages of the grid system shows most. If you make a mistake with the movement codes, the player will find that they could move south to a room, and then move north again, to find that the old room has magically disappeared and been replace by another one, which is rather frustrating. mistakes in the movement codes using the grid system are more likely to be detected, and can be fixed up by the programmer before anyone else plays the game.

Now you have finished the map, you can "play" your game - not on the computer of course, but on paper and in your mind. Wander around the locations, pick up any objects and try to complete the task you have set. You might discover that you have left out a verb that you need, or that you have hidden the key behind the impenetrable hedge, but you can't cut through it because the whipper snipper is locked in the shed. It is much better to find this at an early stage such as this, where the mistake can easily be fixed, rather than discover it when the program is nearly up and running, where at worst you could have to redesign some of your plot and a lot of your program to remove the error.

One problem that occurs with adventures, even for some of the larger and more expensive computers, is what governs the items the player can carry. Some allow the player to carry as many things as he or she likes, but often the number of items is unrealistic unless the player has about seventeen arms. Others allow the player to carry a set number of items. The problem with this becomes apparent if we consider an example from the demonstration adventure with this set of articles. Imagine that the player is only allowed to carry three things. This means that the player couldn't carry a key, a tape, a roll of sticky tape and a book at the same time, but could carry a large and very heavy rock, a fridge and a washing machine!

There are at least two possible solutions to this problem: -

- Provide the player with, or allow the player to find early on, something in which anything found can be carried. A bag, a box, a backpack or a wheelbarrow are all suitable.
- Give each object a weight, and limit the weight a player can carry, rather than the number of objects.

The weight limit could either be fixed, or be based on the strength of a player (if you decide to have a strength rating in your game) at the time an attempt is made to pick up an object. There is no need to use a particular scale, like kilograms or ounces. Just rate your lightest object(s) as 1 unit, and if you think something is roughly 3 times as heavy, rate it as 3 units and so forth. Your scale doesn't need to be particularly accurate as long as it is reasonable.

In the last paragraph, I mentioned a strength rating. Although I won't be writing a great deal on how to program characters for role playing games, these are some of the attributes you might like to include if you decide to create a role playing adventure.

STRENGTH: This is by far the most important attribute in an adventure game, and one you should include even if you aren't writing a role playing adventure. Not only does it govern how much a player can carry, but it also indicates the player's general state of health. If the strength rating reaches zero, it's the end of the line for the player. When the player does something silly, like drink something poisonous, points can simply be deducted rather than respond "SORRY YOU'RE DEAD" each time a wrong action is carried out. Of course there will be times that the player will be killed independant of the strength rating, if he or she happens to fall off a ninety-nine foot cliff, for example. As time goes on, particularly if the player is carrying a lot of heavy things, the strength rating may drop due to tiredness. This can be cured by leaving some food around for the player to eat, or somewhere to sleep. (Originally the players state of health was indicated by a separate HEALTH rating, but both are now usually covered by the STRENGTH rating.)

SKILL: In role playing board games, this is important for it determines how well a blow

is aimed, while strength determined how hard the monster is hit. In computer games, this rating could be also used to decide the result of an action where the player has to show some form of coordination, like firing a gun or swimming, for example. (Ambitious programmers could even include a short arcade game-type test of skill before the start of the game to give a representative skill rating.)

HEIGHT and WEIGHT: The main uses of these are to determine whether players are big enough to carry certain objects (irrespective of strength), or if they can cross dangerous territory, or hide somewhere. Very few hobbits can use six foot swords, but they would have an advantage over giants when crossing rotten wood bridges or hiding in hollow logs.

INTELLIGENCE: This is mostly used to determine the character type of a player during character generation, with wizards being generally more intelligent than barbarians. It also decides if the player can learn new skills during the game.

MAGIC ABILITY: The name says it all. Some character types will be able to do magic, and others won't.

WEALTH: Once again the name says it all. This is used not only to indicate a player's success, but also to allow the player to buy things he or she might need during the game.

LUCK: This can be used to overcome situations when the player might otherwise be killed. One good method of testing a player's luck that I came across chooses a random number between one and ten, and compares it to the player's luck rating. If it is lower than the luck rating, the player survives and the luck rating is decreased by on e point. If not, it's the end for the player. Luck is only tested when the player does something stupid, and it gives the player a chance to survive.

There are several methods of generating a character.
- Have preset values for each attribute. This makes the game, along with non-role playing adventures useful only until it is completed, and then it is "dead." however this is by far the easiest situation to program, and if your program is good enough, this shouldn't matter.
- Have several character types, each with their own attribute values. This gives several different situatons for your game.
- Have the same situation as above, also give the player a number of modification points. with which (s)he can add points to the various ratings. Also allow the player to take some points off some ratings and add these to others. With this method you would also need a subroutine which prevents particular ratings going above or below maximum or minimum values, and another to change the character types if the ratings are changed drastically.
- Have all ratings randomly generated.

Before you are ready to type in the program, you should have all the computer responses to your verbs and nouns prepared. Anything the computer doesn't understand will be covered by a few different "I DON'T UNDERSTAND 'SMASH'" or "TRY SOMETHING ELSE" messages. Just make sure you make it clear what it is that the computer doesn't understand. There should also be a general message for situations where "YOU DON'T HAVE THE BUCKET" (or anything else for that matter). Not only should you have messages for verb-noun combinations that are essential for the game (all the "OK" and "YOUz TOOK THE GOLD" messages), but also for the more irrelevant or silly combinations, you make equally silly replies - for "SWING ROPE" you could reply "THIS IS NO TIME TO PLAY GAMES!" There is no reason why you shouldn't have as much fun writing the game as others have playing it! You might also like to have some interesting replies to the coarse language that some adventurers enter (Tsk Tsk) when they are finding the going hard.

You should also be composing your room descriptions. These should contain all the

information the player needs to know about the room (except for the exits and the visible objects which are covered elsewhere, and anything the player needs to LOOK for).

Once again, as with most other aspects of adventure game design, I can't tell you exactly what to do here .Check the room descriptions (or anything else for that matter) of other adventures but remember that most players prefer original adventures. You should be able to get a vague idea of the lengths that you should have for each description and perhaps the style.

If you want to have longer descriptions the text really starts to chew up the memory. You may be able to squeeze longer descriptions in using a method of tokenising, which stores commonly used words or phrases in the data statements as single character tokens (the inverse character set), in much the same way that programs are stored in BASIC. This allows for much longer descriptions in the available memory, but looking for common words, assigning a token for each and using these in the data statements is, to say the least, rather tedious. You may still think it is worthwhile because of the atmosphere a well written, but slightly longer, room description can add to a game. This method will be discussed in more detail in a later edition. Even if you do use this method, you must limit your descriptions to a length of 255 characters or less. Not only will longer descriptions leave little room on the screen for possible exits, visible objects, and a strength (or any other) rating, but they will also generate a "STRING TOO LONG ERROR"

if you aren't going to use this method, you should note that most room descriptions can begin with the words "You are"; followed by "In", "On", "At", "Near", or "By", then followed by "A","An","The" or "Some". You can therefore leave the "You are" out of the description and add it later on in the program, and substitute the next two words with a number, as was done in the demonstration program. If the third or fourth word of some descriptions are not one of those above, use a code for a blank - "" - instead.

This just about brings to an end the discussion of designing your program. You should now be ready to start designing the program code. Yes, it's more planning!
"I don't need to plan my programs," you might say. However, even if you can write all your other programs without planning, if you try this with an adventure, you will probably leave things out and make the program a tangled mess. Also you will have difficulty tracking down bugs (the ones that don't cause error messages) because you have forgotten exactly where you put that particular section, or that if you want to add some extras to the program, like sound effects, you might find that you have run out of line numbers.

If you think that means you have to draw a flowchart, don't bother. I tried once, and it quickly began to resemble a bowl of spaghetti. Even if you manage the flowchart, it may be difficult to follow, or you could have a nervous breakdown when you discover for the forty second time that you left something out, and because there isn't any room for it, you have to throw it away and start again. You should instead split the program into manageable modules and plan each one separately. these are: -
LOAD ANY MACHINE LANGUAGE ROUTINES INTO MEMORY.

BRANCH TO INITIALISATION SUBROUTINE: (The initialisation is placed at the end of the program because each time the program interpreter encounters a GOTO statement it goes back to the start of the program and looks through all the line numbers until it finds the right one. As this is only used once, considerable time is saved by placing it at the end of the program.)

DISPLAY THE CURRENT LOCATION, EXITS, ETC.

ACCEPT PLAYER INPUT AND PROCESS IT INTO VERB AND NOUN.

DEAL WITH ANY INPUT THAT THE COMPUTER DOESN'T UNDERSTAND.

GREAT FLOCK OF IF... THEN STATEMENTS BRANCHING TO VARIOUS VERB ROUTINES: (or if you have an Extended BASIC - some ON - GOSUB statements. Don't ask me why the designers of

the VZ decided to mask this command when it was already present in the ROM.)

DEAL WITH ANY EVENTS THAT HAPPEN INDEPENDANT OF THE PLAYER'S ACTION: (For example, check to see if the player has died.)

SCREEN DEALING WITH THE PLAYER QUITTING OR DYING.

SUBROUTINE FOR PLAYER MOVEMENT.

INDIVIDUAL SUBROUTINES FOR ALL OTHER VERBS.

INITIALISATION: (Dimensioning arrays, etc)

DATA STATEMENTS: (These are placed at the end of the program for the same reason as the initialisation.)

Here are just a few extra hints for typing in your program. Structural programming enthusiasts will probably lynch me for saying a few of these, but our main concern is memory, and not readability.

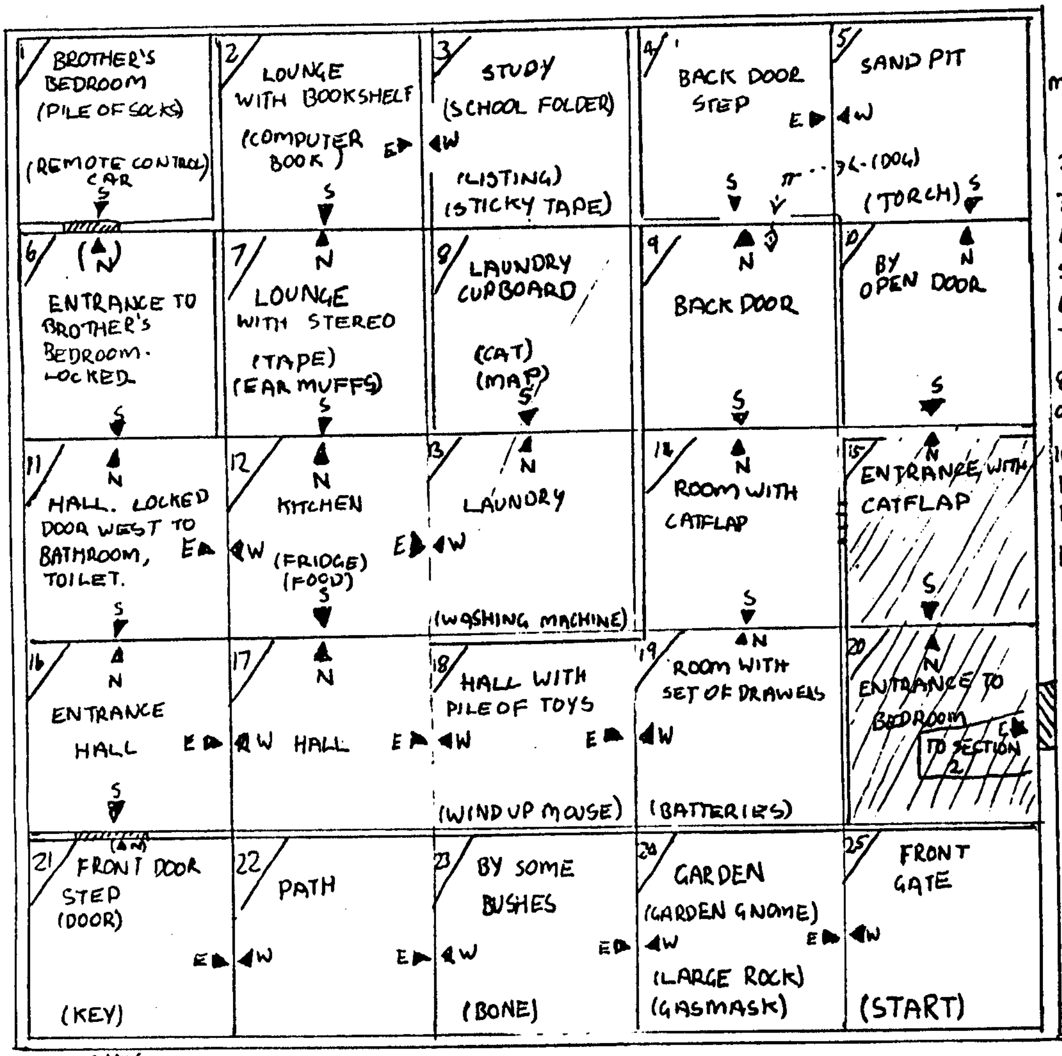
- Use multiple statement lines: This saves memory as each line takes five bytes just to exist. Somewhere around half - perhaps even more - of the lines in an adventure are IF... THEN statements so if you don't use them you will have to waste more memory repeating the condition several times, or have a program that leapfrogs all over the place (which structural programmers probably dislike even more.)
For example:
```
1790  IF(H=7562 OR H=7662)AND F(44)>0 AND C(1)=0 THEN R$="HE TAKES IT": F(64)=1: F(44)=
f(44)-1: RETURN
```
If you didn't use a multi - statement line here you would have to list the long set of conditions many times.

- If you use REM statements, Don't GOTO or GOSUB directly to that line: You will want to remove these eventually, either because you are running low on memory, or because you have finished the program and you don't want to make it too easy for peeping adventurers to cheat! When you do this, you don't need to mess around changing lines in order to avoid "UNDEF'D STATEMENT ERRORS."

- Leave out LETs, unnecessary spaces, etc: Not only do these take up extra memory, but they also take up space in the 64 characters you can have for each line, meaning you have to start new lines which waste even more memory.

In the next few editions, we will look at adding a load/ save feature to your game, and the tokenised room descriptions mentioned earlier, then we will examine each section of the program in detail.

SECTION ONE

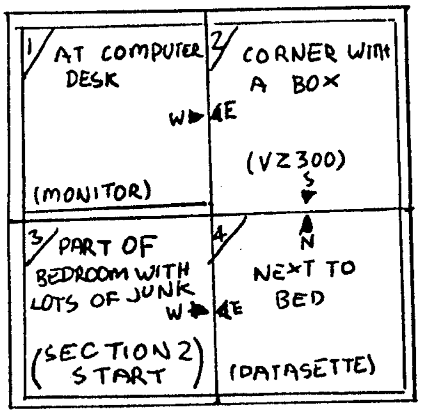


YOU NEED A LIGHT TO SEE IN HERE

MOVEMENT CODES
1) 1011
2) 1010
3) 1101
4) 1010
5) 1001
6) 0011
7) 0011
8) 1011
9) 0011
10) 0011
11) 0010
12) 0000
13) 0101
14) 0011
15) 0011
16) 0010
17) 0100
18) 1100
19) 0101
20) 0110
21) 0110
22) 1100
23) 1100
24) 1100
25) 1101

VERBS	NOUNS
N S W E	LISTING
HELP	MAP
INV	TAPE
GO	BOOK
GET	FOOD
EXAMINE	STICKY TAPE
LOOK	GASMASK
DROP	MOUSE
LEAVE	CAR
OPEN	EARMUFFS
READ	BONE
UNLOCK	KEY
LIGHT	BATTERIES
UNLIGHT	TORCH
EXTINGUISH	ROCK
DO	FRIDGE
WIND	WASHING MACHINE
DIG	DOG
GIVE	CAT
WEAR	FOLDER
KICK	STEREO
EAT	BOOKSHELF
START	BROTHER
PLACE	HOMEWORK
DRIVE	BUSHES
LOAD	DOOR
SAVE	DOOR MAT
QUIT	SOCKS
	DRAWER

SECTION TWO



MOVEMENT CODES
1) 0020
2) 0401
3) 0040
4) 2003

VERBS	NOUNS
N S W E	LIST
HELP	MAP
INV	TAPE
GO	BOOK
GET	FOOD
LEAVE	STICKY TAPE
DROP	VZ300
TYPE	DATASETTE
EXAMINE	MONITOR
LOOK	JUNK
EAT	BOX
LOAD	BED
SAVE	PROGRAM
QUIT	

LET'S INVESTIGATE SOUND ON THE VZ.

PART II

by Bob Kitch.

Last time we produced the Star Wars Theme using a few simple BASIC commands. Let's analyze this action a little further and see what insight this gives us. Recall that there were a few shortcomings or limitations in using the SOUND command.

SOUND COMMAND.

Quite a lot happens in the BASIC Interpeter when the SOUND command is used.

As the Command Interpeter is scanning through a line in the BASIC Program Statement Table, it is searching for the SOUND token 9EH. Upon finding this token, it transfers control to the Verb Action Routine for the SOUND command located at 2BF5H to 2C71H. Perhaps you would like to Disassemble this portion of your ROM and decode it? This section of code looks for the pitch and duration values in the Program Statement Table that form part of the command.

Remember that these values must be in the range of 1 to 31 and 0 to 9 respectively. The routine mentioned above, checks these values. Where does it pick up the correct frequency and duration to pulse the speaker?

Two tables of values are embedded in the ROM. A Frequency Table occurs from 02CFH to 030CH. These are two-byte entries and correspond to the notes A2 to D#5. (31 notes and 62 bytes.) At 0361H to 037FH another 31 byte table exists for the 31 tones. These values correct the duration for the frequency value read from the larger table.

The Verb Action Routine then calls a couple more subroutines in ROM to switch bits 0 and 5 in the Output Latch at 6800H. These are the "low-level" routines that control the piezo speaker. We are now very close to the hardware of the VZ.

The low-level code consists of three subroutines commencing at 3450H to 3483H in ROM. Perhaps some more Disassembly would be illuminating at this point? The main routine is at 345CH to 3468H. On entry, the HL register contains the frequency, and the BC register contains the duration. The entry point from the Verb Action Routine is at 3469H to 3483H where the "cycling" of the Output Latch occurs to achieve the sound. A third BEEP routine occurs from 3450H to 345BH. This section sets the HL register to 160 and the BC register to 6 to provide a beep. This sound is heard every time a key is pressed on the keyboard - so this routine is called quite often.

SOUND EFFECTS PROGRAM.

To illustrate some of these ideas, the accompanying program is useful. A series of sound effects are generated by manipulating the HL and BC registers of the Z80 and by calling the sound routine located

at 345CH in ROM. Note that this is a distinctly different way of making a noise compared to the SOUND command in BASIC. Furthermore, note the variety of sounds that can be produced. The Sound Effects program is more "interesting" than the previous Star Wars melody.

The Sound Effects program uses the USR() statement to connect the BASIC program to the ROM calls. It is an extremely useful and powerful technique. How many of you are familiar with it? It is poorly explained in the VZ Manuals. The 12 bytes of machine code are set out in lines 130 to 180.

The program is also set out to illustrate the "looping" that occurs to place certain values into the HL and BC registers. This form of coding clarifies the procedures.

Next time we will discuss directly switching the latch at 6800H using our own machine code routine - not the one's in ROM. Notice how we are "getting closer to the hardware" and obtaining more control over the sound output.

```
'                ************************
'                **   SOUND EFFECTS    **
'                **     LE VZ  #10     **
'                **  R.B.K. 30/1/86    **
'                **  ALTERED 26/8/90   **
'                **RBK,ANDREW WILLOWS**
'                **KEN CLARKE (NZ)     **
'                ************************

90 '         *****POKE IN M/L ROUTINE*****
100 FOR T%=-28687 TO -28676     :'POKE INTO 8FF1H TO 8FFCH.
110    READ D%:POKE T%,D%
120 NEXT T%
121 PT%=-28685                  :'ADDR. FOR PITCH (L-REG)
122 DR%=-28682                  :'ADDR. FOR DURATION (C-REG)
130 DATA 229                    :'  PUSH HL
140 DATA 033,160,000            :'  LD HL,00A0    PITCH 160
150 DATA 001,003,000            :'  LD BC,0003    DURATION 3
160 DATA 205,092,052            :'  CALL 345C
170 DATA 225                    :'  POP HL
180 DATA 201                    :'  RET
190 POKE30862,241:POKE30863,143:'SET 788E/FH TO F1/8F FOR USR().
200 '*****MAIN MENU*****
210 CLS
220 PRINT:PRINT"          SOUND EFFECTS  ":PRINT
230 PRINT"A DEC ZOOP      B INC ZOOP      ";
240 PRINT"C RANDOM BEEPS  D WAVES         ";
250 PRINT"E INC PHASOR    F DEC PHASOR    ";
260 PRINT"G UFO LEAVING   H UFO LANDING   ";
270 PRINT"I BUZZER        J SHIP SIREN    ";
280 PRINT"K BURGLAR ALARM L POLICE SIREN  ";
```

```
90 PRINT"NB TELEPHONE        NB                    ";
20 PRINT@466,"      ";
30 PRINT@448,"ENTER OPTION #  ";:INPUT OP$
40 POKE DR%,3                        :'SET DURATION TO 3 ON ENTRY.
)0 '*****BRANCH TO CHOICE*****
.0 IF OP$="A",1100 ELSE IF OP$="B",1200
20 IF OP$="C",1300 ELSE IF OP$="D",1400
50 IF OP$="E",1500 ELSE IF OP$="F",1600
40 IF OP$="G",1700 ELSE IF OP$="H",1800
50 IF OP$="I",1900 ELSE IF OP$="J",2000
)0 IF OP$="K",2100 ELSE IF OP$="L",2200
'0 IF OP$="M",2300 ELSE IF OP$="N",210
0 GOTO320
00 '*****DECAYING ZOOP*****
10 CLS:PRINT@232,"DECAYING ZOOP"
20 FOR T%=1 TO 255 STEP 4     :'LOWER PITCH - FIXED DURATION.
30     POKE PT%,T%
40     X=USR(0)
50 NEXT T%
60 GOTO 210
00 '*****INCREASING ZOOP*****
10 CLS:PRINT@232,"INCREASING ZOOP"
20 FOR T%=255 TO 1 STEP -4    :'RAISE PITCH - FIXED DURATION.
30     POKE PT%,T%
40     X=USR(0)
50 NEXT T%
50 GOTO 210
)0 '*****RANDOM BEEPS*****
.0 CLS:PRINT@234,"RANDOM BEEPS"
'0 POKE DR%,10                  :'CONSTANT DURATION.
50 FOR Y%=1 TO 50               :'DO 50 REPETITIONS - VARY PITCH.
.0     T%=RND(254)+1
)0     POKE PT%,T%
0      X=USR(0)
0 NEXT Y%
0 GOTO 210
0 '*****WAVES*****
0 CLS:PRINT@237,"WAVES"
5 POKE DR%,1                    :'FIX DURATION.
0 FOR Y%=1 TO 10                :'DO 10 OSCILLATIONS.
5    FOR T%=1 TO 10             :'LOWERING PITCH.
)        POKE PT%,T%
5        X=USR(0)
)    NEXT T%
5    FOR T%=30 TO 1 STEP -1 :'RAISING PITCH.
)        POKE PT%,T%
5        X=USR(0)
)    NEXT T%
5 NEXT Y%
) GOTO 210
) '*****INCREASING PHASOR*****
 CLS:PRINT@230,"INCREASING PHASOR"
 FOR Y%=20 TO 1 STEP -1     :'INCREASE STARTING PITCH.
    FOR T%=Y% TO 1 STEP -1  :'DECREASE NUMBER OF TONES.
        POKE PT%,T%

        X=USR(0)
    NEXT T%
```

```
1570 NEXT Y%
1580 GOTO 210
1600 '*****DECREASING PHASOR*****
1610 CLS:PRINT@230,"DECREASING PHASOR"
1620 FOR Y%=1 TO 20                   :'DECREASE STARTING PITCH.
1630    FOR T%=1 TO Y%                :'INCREASE NUMBER OF TONES.
1640        POKE PT%,T%
1650        X=USR(0)
1660    NEXT T%
1670 NEXT Y%
1680 GOTO 210
1700 '*****UFO LEAVING*****
1710 CLS:PRINT@233,"UFO LEAVING"
1720 T%=61                            :'SET PITCH.
1730 FOR D%=60 TO 1 STEP -1           :'DECREASE DURATION.
1740    POKE DR%,D%
1750    POKE PT%,T%
1760    T%=T%-1                       :'LOWER PITCH.
1770    X=USR(0)
1780 NEXT D%
1790 GOTO 210
1800 '*****UFO LANDING*****
1810 CLS:PRINT@233,"UFO LANDING"
1820 T%=1                             :'SET PITCH.
1830 FOR D%=1 TO 60                   :'INCREASE DURATION.
1840    POKE DR%,D%
1850    POKE PT%,T%
1860    T%=T%+1                       :'RAISE PITCH.
1870    X=USR(0)
1880 NEXT D%
1890 GOTO 210
1900 '*****BUZZER*****
1910 CLS:PRINT@236,"BUZZER"
1920 POKE DR%,3
1930 POKE PT%,60
1940 FOR Y%=1 TO 100                  :'SOUND TONE 100 TIMES.
1950    X=USR(0)
1960    FOR D%=1 TO 5                 :'SMALL DELAY.
1970    NEXT D%
1980 NEXT Y%
1990 GOTO 210
2000 '*****SHIP SIREN*****
2010 CLS:PRINT@234,"SHIP SIREN"
2015 POKE DR%,8                       :'FIX DURATION.
2020 FOR Y%=1 TO 10                   :'DO 10 REPEATS.
2025    FOR T%=200 TO 80 STEP -8:'INCREASE TONE.
2030        POKE PT%,T%
2035        X=USR(0)
2040    NEXT T%
2045    FOR D%=1 TO 150               :'DELAY BETWEEN REPEATS.
2050    NEXT D%
2055 NEXT Y%
2060 GOTO 210
2100 '*****BURGLAR ALARM*****
2110 CLS:PRINT@233,"BURGLAR ALARM"
```

```
2120 POKE DR%,255                :'FIX DURATION TO MAX.
2130 FOR Y%=1 TO 5               :'DO 5 CYCLES.
2140    POKE PT%,50              :'TONE HI.
2150    X=USR(0)
2160    POKE PT%,60              :'TONE LO.
2170    X=USR(0)
2180 NEXT Y%
2190 GOTO 210
2200 '*****POLICE SIREN*****
2210 CLS:PRINT@233,"POLICE SIREN"
2215 POKE DR%,9                  :'FIX DURATION.
2220 FOR Y%=1 TO 5               :'DO 5 REPEATS.
2225    FOR T%=200 TO 80 STEP -4
2230       POKE PT%,T%           :'FAST RISING PITCH.
2235       X=USR(0)
2240    NEXT T%
2245    FOR T%=80 TO 200
2250       POKE PT%,T%           :'SLOW FALLING PITCH.
2255       X=USR(0)
2260    NEXT T%
2265 NEXT Y%
2270 GOTO 210
2300 '*****TELEPHONE*****
2310 CLS:PRINT@234,"TELEPHONE"
2315 POKE DR%,15                 :'FIX DURATION.
2320 FOR Y%=1 TO 5               :'DO 5 REPEATS.
2325    FOR D%=1 TO 2            :'SOUND DOUBLE RING.
2330       FOR T%=1 TO 8         :'DO 8 WARBLES.
2335          POKE PT%,100       :'LO TONE.
2340          X=USR(0)
2345          POKE PT%,50        :'HI TONE.
2350          X=USR(0)
2355       NEXT T%
2360       FOR T%=1TO 50         :'PAUSE BETWEEN DOUBLE RINGS.
2365       NEXT T%
2370   NEXT D%
2375   FOR D%=1 TO 400           :'PAUSE BETWEEN REPEATS.
2380   NEXT D%
2385 NEXT Y%
2390 GOTO 210
10000 CLS:PRINT"ERASING SOUNDS":ERA"SOUNDS"
10010 PRINT"SAVING SOUNDS":SAVE"SOUNDS"
10020 END
```

HACKERS AND PIRATES.

This information was sent in by Ben Hobson, who thinks it may be of interest to others.

I have found the problem with SPRITE GENERATOR crashing things. In my case DOS.

I have not tried it, but possibly relocate DOS or Extended Basic directly after the SPRITE routines, as maybe Sprite Generator uses Hi-Mem, or it may alter IX or IY registers. Therefor a complete rewrite of Sprite Generator or of EXT. BASIC may be needed.

TRADING POST

EPROMS for EXTENDED DOS. and BASIC

Are available from

Bob Kitch
7 Eurella Str.
KENMORE Q'ld. 4069

FOR SALE

PRINTER GP 100. VZ compatiable. New ribbon. $100.
Apply Editor.

TO SWAP: VZ 300 PCB (no case or keyboard), Joystick interface with 1 joystick, 9 x 6116 2k RAM chips, Z80A, 2716 EPROM, 2 x 482764 EPROMs, Plus various ICs, eg 74ls138, 74ls157. All chips are socketed (except Z80A, EPROMS) and are from a defunct microbee. (The VZ300 does not work but only needs a U14 i.c. Perhaps a good one may be desoldered from one of the old VZ300s the Editor has for sale.
FOR: VZ200 in working order. (with or without keyboard)
If this does not please you PLEASE contact me anyway as I have many other goodies I will include to make the deal.
Ben Hobson , P.O. BOX 255, QUIRINDI, 2343 or phone (067) 462076 after 4pm.

***

OTHER V Z USER GROUPS

H.V.V.Z.U.G
P.O.Box 161
JESMOND NSW.2299.

DISKMAG
P.O.Box 600.
Taree NSW. 2430.

CENT.VIC.COMP.Club
24 Breen St.
BENDIGO VIC 3550

BRISBANE VZUG
63 Tingalpa St.
WYNUM West. Q'ld. 4178

Graeme Bywater
P.O.Box 388
Morley W.A. 6062



"What a day! The computer broke down and we all had to think!"

it
ow
of
th
ey
ng.
un
T
ntl
ode
V
it
n
ne
pre
es
lig
nd
e i
HOM
iss
I w
Al
uip